Universidade Federal de Pernambuco (UFPE)
Centro de Informática (CIN)
Mestrado Profissionalizante em Ciência da Computação

Orientadora: Profª Carina Alves

Utilização da Metodologia de Desenvolvimento Orientado a Aspectos no Mercado Local

Por

Edson Almeida

Eduardo Carneiro

Edvaldo Panta

Roberto Tabosa

Recife, Junho de 2007

Universidade Federal de Pernambuco (UFPE)
Centro de Informática (CIN)
Professional Master in Computer Science

Advisor: Teacher Carina Alves

Using the methodology of Aspect-Oriented Development in the local market

By

Edson Almeida

Eduardo Carneiro

Edvaldo Panta

Roberto Tabosa

Recife, June of 2007

Sumário

SUMÁRIO III

RESUMO IV

ABSTRACT IV

1. INTRODUÇÃO 5

1.1 Motivação 5

1.2 Objetivo 5

2. ASPECT-ORIENTED PROGRAMMING 6

2.1 Filosofia 7

2.2 Object-Oriented Programming 8

2.3 Subject-Oriented Programming 11

3. FUNCIONAMENTO DA AOP 13

4. UTILIZAÇÃO DE AOP NO MERCADO LOCAL 22

5. CONCLUSÃO 23

6. REFERÊNCIAS BIBLIOGRÁFICAS 24

Resumo

A Programação Orientada a Aspectos ou AOP (Aspect-Oriented Programming) é um novo paradigma que surge para o desenvolvimento de softwares. A despeito de outras metodologias de implementação de software, a Programação Orientada a Aspecto é pouco difundida no mercado. Este trabalho apresenta o resultado de um estudo efetuado junto às empresas desenvolvedoras de software no mercado local de Recife. Mapeamos e descrevemos neste documento as dificuldades e motivações para a adoção da AOP no mercado local para melhoria da produtividade do desenvolvimento de sistemas.

Palavras-chaves: OOP (Object Oriented Programming); AO (Aspect Oriented), SOP (Subject Oriented Programming); Mercado local.

Abstract

The Aspect-oriented programming or AOP, is a new paradigm that comes to the software's development. In spite of other methods of software's implementation, the aspect-oriented programming is little known in the market. This paper presents the results of a study done in the software's developers companies on the local market of Recife. We mapped and described in this document the difficulties and motivations for the use of AOP in the local market to improve the productivity of development's system.

Key words: Object Oriented Programming, Aspect Oriented, Subject Oriented Programming and Local Market.

1. Introdução

Este artigo abordará os principais assuntos referentes ao paradigma da programação orientada a aspectos, bem como apresentará uma abordagem prática da utilização dessa metodologia por empresas desenvolvedoras de software instaladas na Região Metropolitana de Recife.

1.1 Motivação

O fator mais importante para escolha do tema é, inicialmente, aprofundar conhecimentos na programação orientada a aspectos e delinear a utilização dessa metodologia por parte das empresas desenvolvedoras de software do arranho produtivo local.

1.2 Objetivo

Realizar o diagnóstico de empresas que tenham interesse na implantação da metodologia de desenvolvimento orientado a aspectos no mercado local da região Metropolitana do Recife.

2. Aspect-Oriented Programming

As metodologias e ferramentas para o desenvolvimento de software têm sofrido, ao longo dos anos, avanços muitas vezes imensuráveis em forma e conteúdo, trazendo benefícios para empresas desenvolvedoras e clientes finais. Evolução essa, que busca alguns objetivos comuns, como: redução do número de linhas de código, re-uso de código, redução do custo de manutenção e independência de código.

Por questões culturais, a cada nova metodologia proposta, o meio acadêmico é o primeiro a tratar o assunto e submetê-lo a testes, verificações, ajustes e propostas. Porém, a despeito do que acontecia há décadas atrás, o mercado, aqui na figura das empresas desenvolvedoras de sistemas, está cada vez mais absorvendo e se adaptando com menor dificuldade a essas maneiras inovadoras de implementações e, quando isso acontece em ampla cadeia de um segmento, temos aí o chamado *paradigma* (modelo, padrão) para o desenvolvimento de software.

Uma das metodologias que está se tornando um paradigma para desenvolvimento de software é a Aspect-Oriented Programming (OAP) ou Programação Orientada a Aspectos. A OAP foi sugerida, por Gregor Kickzales, em 1996, como uma série de diretrizes agregadas a OOP para facilitar o desenvolvimento de software.

2.1 Filosofia

A AOP é uma alternativa com o propósito de resolver problemas que não são facilmente solucionados com OOP, nem com a programação estruturada. Ela está baseada no conceito de aspectos, desta forma para um melhor entendimento da AOP será necessária uma breve abordagem sobre aspectos.

O que são aspectos?

Aspectos são características adicionais de um sistema, ou seja, são funcionalidades extras que aumentam o nível de complexidade de um sistema, não apenas características específicas de um componente.

A AOP foi criada pela Xerox em 1997, com o objetivo de construir uma abordagem a fim de permitir que a linguagem de programação pudesse expressar da melhor maneira as características sistemáticas do comportamento da aplicação, onde essa teria uma linguagem central, porém diversas linguagens de domínio. Essas características também são conhecidas como ortogonais ou transversais.

Inicialmente os compiladores que trabalhavam com o AOP através do conceito de meta-programação, onde era necessário ser gerado um código à parte o qual eram acrescidos os elementos para dar suporte às novas abstrações, ou seja, um código que gera como resultado outro código, em seguida é novamente compilado para gerar o produto final. Outra característica do AOP é o conceito de reflexão computacional, onde parte do código gerado é utilizada para fazer alterações na própria aplicação.

A AOP é uma alternativa com o propósito de resolver problemas que não são facilmente solucionados com OOP, nem com a programação estruturada. Ela está baseada no conceito de aspectos, introduzindo novas abstrações, desta forma para um melhor entendimento da AOP será necessária uma breve abordagem sobre aspectos, o que iremos abordar mais adiante.

Crosscutting

Através da utilização da AOP, os interesses comuns do sistema são separados pelo programador, esse comportamento, porém não é feito através de módulos, como na OO, onde a unidade de modularização é uma classe. Ele é conhecido como um comportamento que atravessa(**crosscutting**) o sistema e está espalhado em diversas classes. Desenvolvedores que utilizam o conceito de crosscutting geralmente possuem problemas como a falta de modularidade. É onde entram os aspectos, ou seja, para manter a consistência do projeto, através de um novo nível de modularidade.

Outras metodologias trazem abordagens diferentes ou mesmo, são re-utilizadas e adaptadas aos interesses almejados. Destacam-se aqui a já conhecida Object-Oriented Programming (OOP) e a Subject-Oriented Programming(SOP).

2.2 Object-Oriented Programming

A Programação Orientada a Objeto (Object-Oriented Programming) é o paradigma atual de programação. Foi concebida na década de 70 na linguagem SIMULA-68, posteriormente na linguagem que popularizou essa metodologia, a Smalltalk, sendo que a linguagem Java difundiu essa metodologia no meio acadêmico e comercial. O principal objetivo ao se criar a Programação OO foi à redução de custos com manutenção, através da reutilização das linhas de código e modularidade da escrita.

A OOP permite criar componentes de forma a separar as partes do sistema por responsabilidades. Essas partes comunicam-se entre si através de mensagens. Esses componentes são chamados de Objetos.

Um Objeto na OOP é *uma unidade dinâmica, composta por um estado interno privativo (estrutura de dados) e um comportamento (conjunto de*

operações).

Os Objetos, por sua vez, possuem duas características em comum:

Estado revela seus dados importantes, por exemplo, Objeto pessoa possui: nome, cor, idade etc. É mantido nas chamadas variáveis.

Comportamentos são as ações que o objeto pode executar, por exemplo, uma pessoa pode falar, correr etc. É implementado através dos chamados Métodos.

Ao conjunto de Objetos de mesmo tipo, com Estado e Comportamento em comum denominamos Classe.

Tão importantes quanto os objetos para a OO são a comunicação e interação entre os mesmos que se dá quando da execução de um Método.

Uma linguagem OO possui alguns recursos que a caracterizam como tal:

Herança é um recurso em que um conjunto de instâncias(objetos) são criadas a partir de uma outra instância, possuindo mesmas características, ou seja, novas classes podem ser definidas a partir de uma classe já existente. É um recurso que permite alta reutilização de código.

Polimorfismo ("várias formas") é um recurso que permite que duas ou mais classes criadas a partir de uma superclasse, possuam métodos com comportamentos distintos, peculiares a cada uma das classes.

Encapsulamento é um recurso que separa os aspectos internos e

externos de um objeto.

Abstração é um recurso/habilidade de focar nos aspectos essenciais de um conceito. Permite modelar características do mundo real.

Como em toda metodologia de implementação, existem várias vantagens que as fazem evidenciar-se, como também desvantagem que abrem espaço para novas tecnologias e métodos a serem aplicados.

Algumas vantagens encontradas com uso da OOP:

- Codificação estrutural e procedural intuitiva;
- Maior facilidade para representar entidades complexas;
- APIs para diversos tipos de linguagens e aplicações;
- Foco na padronização, reuso e extensibilidade.

Algumas desvantagens do uso da OOP:

- Aspectos comportamentais são implementados de maneira procedural;
- Não utilização de técnicas de aprendizagem de máquina para classes e objetos;
- Sem semântica declarativa explícita.

Os recursos da OOP fazem dessa metodologia, sem dúvida, o paradigma atual para codificação de sistemas com características de manutenibilidade que a elevam a um grau investimento elevado em diversos setores do mercado. Porém, como tantas outras tecnologias, é passível de melhorias, como a AOP e a SOA, que além de tendências já passam a fazer parte da realidade das empresas.

2.3 Subject-Oriented Programming

A necessidade de desenvolver software de qualidade aumentou o uso da orientação a objetos [Mayer, Booch] em busca de maiores níveis de reuso e manutenibilidade, aumentando a produtividade do desenvolvimento e o suporte a mudanças de requisitos. Entretanto, o paradigma orientado a objetos tem algumas limitações [Ossher, Harrison], como o entrelaçamento e o espalhamento de código com diferentes propósitos, por exemplo, o entrelaçamento de código de negócio com código de apresentação, e o espalhamento de código de acesso a dados em vários módulos do sistema. Parte destas limitações pode ser compensada com o uso de padrões de projetos [Buschmann, Gamma].

Por outro lado, extensões do paradigma orientado a objetos, como *aspect--oriented programming* (programação orientada a aspectos) [Elrad, Filman], *subject--oriented programming* [Ossher, Harrison], e *adaptive programming* [J., L. K., I., S.-L], tentam solucionar as limitações do paradigma orientado a objetos. Estas técnicas visam obter uma maior modularidade de software em situações práticas onde a orientação a objetos e padrões de projetos não fornecem suporte adequado.

SOP segundo [Ossher, Kaplan, Katz, Harrison e Kruskal] é uma tecnologia de composição de programas que suporta construções baseadas no paradigma de orientação a objetos como composições de sujeitos. Neste conceito, o termo “subjects” caracteriza um conjunto de especificações de estados e comportamentos que refletem a uma concepção do mundo (*identity*).

Objetivos

- Facilitar o desenvolvimento e a evolução de conjuntos de aplicações cooperativas, ou seja, aplicações que compartilham os mesmos objetos e agem em conjunto na execução de operações.

- Possibilidade de desenvolver aplicações separadamente e depois compô-las (juntá-las)
- As aplicações separadas não devem depender umas das outras
- As aplicações compostas podem ter um nível de interação alto ou baixo
- Possibilidade de desenvolvimento de novas aplicações, sem que seja necessário alterar as já existentes, e aproveitando os objetos persistentes já existentes.
- Aplicações não esperadas devem ser suportadas
- Dentro de cada aplicação, deve ser possível utilizar as vantagens da herança, do encapsulamento e do polimorfismo.

Contextualização e aplicações

Subjects podem ser compostos para determinar novas perspectivas, através de regras de: (1) correspondência – utilizadas para especificar correspondências entre conjunto de classes, métodos e atributos de subjects diferentes; e (2) combinação – utilizadas para especificar como atributos e métodos de duas classes serão combinados (sobreposição, ordem de execução) para formar uma nova perspectiva.

Dessa forma é possível expressar evoluções de um conjunto de classes, através da combinação entre o subject que representa este conjunto e um novo subject que realiza adaptações necessárias ao primeiro.

A técnica funciona da seguinte forma:

No momento da compilação, os “subjects” serão compilados primeiro, gerando seus “labels”. Depois, as regras de composição serão aplicadas, criando uma composição que irá funcionar em tempo de execução. Todas as chamadas a um “subject” serão mapeadas para sua composição. Ou seja, a chamada a um método de um “subject” irá, na verdade, chamar a operação na composição.

Conclusões

Subject Oriented Computing é um enfoque baseado na especificação de requisitos cujo objetivo é a síntese de modelos de projeto OO por meio de *design subjects* .

Apesar sua utilidade na descrição de interesses e funcionalidades do sistema -de forma estrutural-ainda falta pesquisa e resultados sobre a parte funcional dos *subjects* ou mesmo ferramentas que apóiem sua representação.

3. Funcionamento da AOP

Para que programadores pudessem resolver problemas como de entrelaçamento e repetição de código, aumentando a reusabilidade e trazendo isso para uma única unidade de implementação chamada aspecto, aumentando a facilidade de manutenção, já que se encontra centralizado em uma unidade não há necessidade da reescrita das classes, dessa forma reduzindo também a complexidade dos componentes, pois o na definição dos aspectos são encontrados partes do código.

Com a diminuição do grau da complexidade, muitos benefícios surgem, pois menos códigos serão escritos, menos erros ocorrerão, o custo de implementação também será menor e a quantidade de programadores conseqüentemente diminuirá. Além de facilidade o gerenciamento da complexidade dos componentes.

Como AOP trabalha em paralelo com outros paradigmas, os conceitos já existentes são reaplicáveis, fornecendo não apenas uma decomposição funcional dos problemas, mas também sistemática, o que permite a separação de requisitos funcionais e não-funcionais através da abstração de aspectos.

Sistemas que possuem vários componentes funcionais não conseguem expressar bem algumas propriedades como sincronização, persistência, interação entre componentes e distribuição, através das notações e linguagens atuais, dessa maneira, essas propriedades precisam ser expressas através de partes de código espalhados nos componentes da aplicação.

Benefícios do AOP

- ✓ Menor quantidade de linhas de código;
- ✓ Maior coesão entre os objetos;
- ✓ Maior legibilidade
- ✓ Maior facilidade de manutenção;
- ✓ Custo e tempo de desenvolvimento reduzido;
- ✓ Alta reusabilidade do código;
- ✓ Menor acoplamento entre os objetos.

Características do AOP

Sincronização: AOP fornece uma forma bastante eficiente para modularização da política de sincronização.

Distribuição: Pode tornar transparente a distribuição através da criação de um aspecto destinado a representar esse interesse, descartando a necessidade de refatoração. Pois o aspecto gera um código (refatoramento) através do mapeamento do interesse de distribuição pelo combinador aspectual.

Tratamento de exceções: AOP centraliza o tratamento de exceções através de unidades elementares, facilitando assim a legibilidade e manutenção das classes para o tratamento de exceções.

Coordenação: AOP modulariza a forma de sincronização da integração entre os objetos ativos na busca do objetivo geral.

Persistência: Existe a possibilidade de centralização e abstração da implementação dos requisitos na camada de negocio, o que irá facilitar uma melhor reusabilidade e manutenção da aplicação..

Combinação aspectual

É o processo da combinação entre a linguagem de componentes e a linguagem de aspectos. É um processo que ocorre antes da compilação, o que gera um código intermediário capaz de executar a operação solicitada durante a execução da aplicação.

A combinação conceitual pode ser dividida em: Estática, onde não há necessidade da existência dos aspectos durante a compilação e execução, trazendo dessa forma agilidade prevenindo que um maior nível de abstração prejudique a performance da aplicação. Ou pode ser Dinâmica, onde será necessária a existência dos aspectos em tempo de compilação e execução, fazendo com que o combinador possa adicionar, adaptar ou remover aspectos durante a execução da aplicação. Uma representação simplificada desse processo pode ser observada na figura 1.

O combinador é responsável por gerar um novo código através da utilização dos aspectos e componentes.

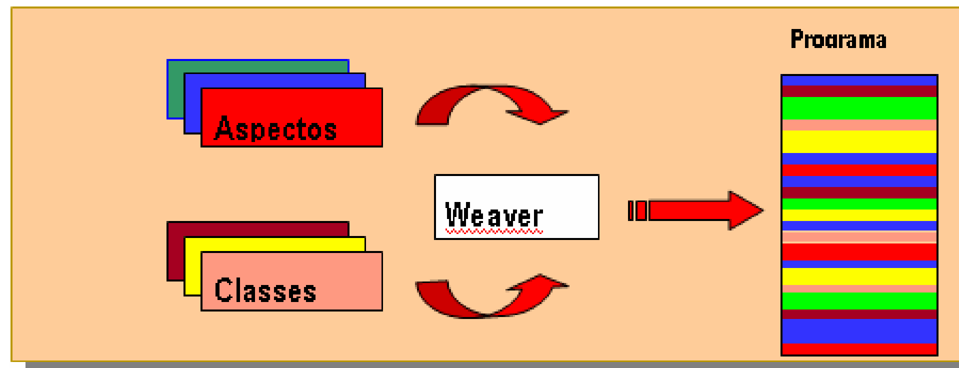


Figura 1: Funcionamento Orientação a Aspectos

Classes: fazem encapsulamendo dos requisitos funcionais.

Aspectos: fazem o encapsulamento dos requisitos transversais.

Weaver: fazem entrelace das classes e dos aspectos do programa.

AspectJ

É um pré-processador weaver que dá suporte a programação orientada a aspectos. É uma extensão do java de fácil aprendizagem e utilização, disponível gratuitamente. Porém algumas preocupações como compatibilidade de linguagem, plataforma, ferramentas e desenvolvedor foram e são levadas em consideração antes de utilizar AspectJ. Pois apesar de utilizar à mesma tecnologia (Java).

Aspecto pode:

- ✓ Possuir tipo;
- ✓ Ser estendido;
- ✓ Ser abstrato ou concreto;
- ✓ Conter campos, métodos e tipos como membros.
- ✓ Conter pointcuts e advices como membros;
- ✓ Acessar membros de outros tipos.

Aspecto não pode:

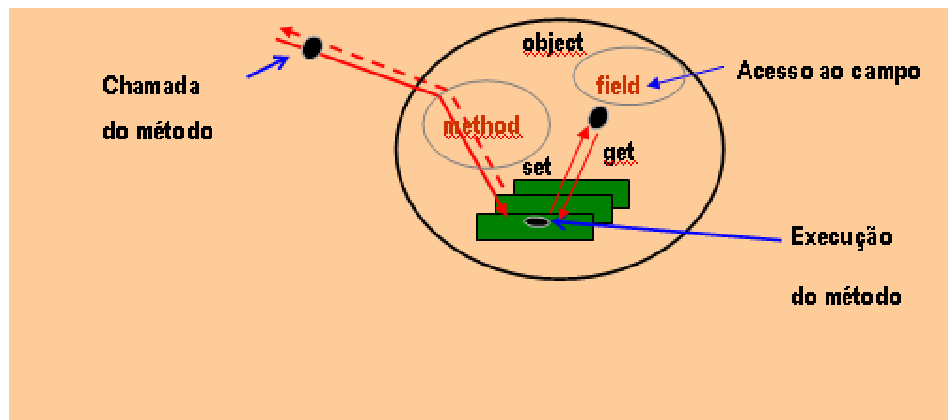
- ✓ Possuir construtor e destrutor;
- ✓ Ser criados com o operador new;

Conceitos da AOP

A programação orientada a aspectos possui quatro conceitos (**join points, advice, pointcut e aspectos**) que serão abordados abaixo.

Joinpoints

São pontos de execução no programa, nos quais os aspectos poderão interceptar as classes, ou seja, permite executar um código adicional antes ou depois da chamada de um método caso haja necessidade.



Modelo de componentes – É um modelo que define princípios, regras e tipo de componentes que facilitam o desenvolvimento das soluções dos problemas.

Os designadores são:

```
call();
execution();
initialization();
handler();
get();
set();
this();
target();
args();
within();
cflow();
```

Pointcuts

São os novos pontos do sistema, onde um cross-cutting pode ser aplicado. Têm como objetivo a criação de regras que irão definir quais eventos serão aplicados pelo join points, sem que haja necessidade de individualizar essas definições. Os pointcuts também precisam apresentar dados de cada execução dos join points para as rotinas que foram executadas.

Quando os join points são identificados, eles são agrupados para um pointcut. Dessa forma é notável a ocorrência dos interesses sistemáticos, pois esses interesses estão em classes distintas na aplicação implementando diversos interesses.

Advice

São partes da implementação de um aspecto que fazem associação aos pointcuts no programa principal, eles incluem um novo comportamento nos join points que fazem parte desse pointcut. Ou seja, são pedaços dessa implementação, executados em pontos estratégicos do programa principal. Ela é divida em: pontos de atuação, onde são definidas as regras de obtenção dos joint points e na parte a qual o código será executado quando o ponto de atuação for chamado. Advice comparada com a OOP é bastante similar à chamada de um método como será visto mais adiante.

Aspectos

É o mecanismo que possui o objetivo de agrupar as partes do código que

referentes aos requisitos não funcionais em uma unidade única na aplicação. Ou seja, são interesses que não são inerentes ao negocio, porém necessários. Como por exemplo, numa autoria da aplicação onde pudesse haver um log de determinados comportamento ou chamadas de métodos de uma classe, ou controle para um determinado tipo de acesso na aplicação. Logo abaixo veremos algumas formas básicas, como é feito o fluxo da informação na AOP e um pequeno exemplo de um Hello World que exemplifica a codificação com AOP.

Tipo:

```
Before();
After();
Around().
```

Formas Básicas

```
before(param) : pointcut(param) {...}
after(param) : pointcut(param) {...}
after(param) returning [formal] : pointcut(param) {...}
after(param) throwing [formal] : pointcut(param) {...}
type around(param) [throws typelist] : pointcut(param) {...}
```

Fluxo da Informação

```
pointcut get(int index) :

 args(index) && call(*get*(..));
before(int index) : get(index) {

        System.out.println(index);

        }
```

Exemplo Hello World

```
public class Hello {
  public static void main(String args[]) {
   System.out.println("Hello world!");
  }
}
```

```
public aspect Greetings {
  pointcut pc() : call(* *main(..));
  before() : pc() { System.out.println("Oi."); }
  after() : pc() { System.out.println("Tchau...");
}
}
```

Saída:

```
>ajc Hello.java Greetings.java
>java Hello
Oi.
Hello world!
Tchau...
```

4. Utilização de AOP no Mercado Local

Foram realizados estudos direcionados a empresas locais do mercado de Recife-PE, através de um questionário, cujo propósito é obter informações sobre empresas desenvolvedoras de produtos de software em Pernambuco. Em especial sobre o desenvolvimento orientado a objetos e aspectos.

Este questionário foi remetido a 30 empresas, destas obtivemos respostas de 10 empresas, onde a partir destas respostas chegamos as seguintes conclusões:

1. O Mercado local tem como padrão o desenvolvimento orientado a objetos;
2. AOP é do conhecimento da maioria das empresas,
3. AOP ainda não é uma realidade como padrão de desenvolvimento;
4. AOP não é utilizada por não ter sido identificado perante estas empresas um ganho que justificasse a adoção deste padrão.

5. Conclusão

Os principais objetivos com este trabalho foram alcançados:

- Entender como a AOP pode ser implementada em ambientes de empresas heterogêneas.

- Estudar a viabilidade de implantação desta metodologia no mercado local

- Avaliar o entendimento desta metodologia pelas empresas de desenvolvimento de software.

Muito do assunto pode ainda ser debatido e relatado, por exemplo, desenvolver um projeto de implementação do paradigma em um projeto numa empresa local buscando identificar as principais dificuldades encontradas na execução do modelo.

Uma melhoria ou sugestão para trabalhos futuros é que o estudo seja aprofundado com a utilização de cases de uso em projetos nas empresas locais. Desta forma poderemos quantificar índices de desempenho para utilização deste paradigma, justificando dessa forma a viabilidade de implantação deste modelo nas empresas de desenvolvimento de software local.

6. Referências Bibliográficas

PRICE, T. **Programa de Especialização:** opção para o Mestrado. Porto Alegre: UFRGS, 1997.

Elrad, T.; Filman, R. E.; Bader, A. **Aspect-oriented programming**. *Communications of ACM*, 44, (10), p. 29-32, 2001.

Mayer, B. ***Object-Oriented Software Construction.*** Prentics Hall, second edition, 1997.

Gosling, J.; Joy, B.; Steele, G.; Bracha, G. **Aspect-oriented programming with AspectJ.** *In OOPSLA´01, Tutorial*, Tampa FL, 2001.

Murphy, G. C.; Walker, R. J.; Baniassad, E. L.; Robillard, M. P.; Lai, A.; Kersten, M. A. **Does aspect-oriented programming work?** *Communications of the ACM*, 44, (10), p. 75-77, 2001.

Buschmann, F.; Meunier, r.; Ronhert, H.; Sommerlad, P.; Stal, M. ***A System of Patterns:*** *Pattern-Oriented Software Architecture*. John Wiley and Sons Ltd, Chichester, 1996.

Filman, R. E.; Friedman, D. P. **"Aspect-Oriented Programming is Quantification and Obliviousness."** Workshop on Advanced Separation of Concerns, OOPSLA 2000. Minneapolis, 2000.

The AspectJ Programming Guide, Xerox Corporation, 002.

Booch, G. ***Object-Oriented Analysis and Design with Applications*.** Benjamin/Cummings, second edition, 1994.

Ossher, H.; Kaplan, M.; Katz, A.; Harrison, W.; Kruskal, V. **Specifying subject-oriented composition**. *TAPOS*, 2, (3), p.179-202. Special Issue on Subjectivity in OO Systems. 1996.

Ossher, H.; Tarr, P. **Using subject-oriented programming to overcome common problems in object-oriented software development/evolution.** In *International Conference on Software Engineering, ICSE´99*, p. 698-688, ACM. 1999.

Buschmann, F.; Meunier, R.; Ronhert, H.; Sommerlad, P.; Stal, M. ***A System of Patterns:*** *Pattern-Oriented Software Architecture*. Chichester, John Wiley and Sons, 1996.

[SOP] Homepage sobre Subject-Oriented Programming Project, IBM J. Watson Research Center, Yorktown Heights, New York, Disponível em: <http://www.research.ibm.com/sop/> Acesso em: 20 mar. 2007.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. ***Design Patterns:*** *Elements of Reusable Object-Oriented Software*. MA, Addison-Wesley. 1994.

Elrad, T.; Filman, R. E.; Bader, A. **Aspect-oriented programming.** *Communications of ACM*, 44, (10), p.29-32. 2001.

Lieberherr, K.J.; Lepe, I. S.; Xiao C. **Adaptive Object Oriented Programming Using Graph-Based Customization.** *Communications of the ACM*, 37, (5), p. 94-101, 1994.

This document was created with Win2PDF available at http://www.win2pdf.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.